UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Diseño e implementación de un sistema de evaluación para formación presencial

Juan Pintado Cort
Tutor: Álvaro Ortigosa Juárez

MAYO 2018

DISEÑO E IMPLEMENTACIÓN DE UN SISTEMA PARA LA EVALUACIÓN PRESENCIAL

AUTOR: Juan Pintado Cort
TUTOR: Álvaro Ortigosa Juárez

Dpto. Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Mayo de 2018

Resumen (castellano)

Este Trabajo Fin de Grado está desarrollado con el fin de crear una aplicación que facilite a alumnos y profesores la tarea de la evaluación presencial, es decir, lo que comúnmente se conoce como pasar lista. Como objetivo secundario, el trabajo trata también de investigar el desarrollo de aplicaciones web con una de las modernas plataformas que unen las últimas tecnologías y frameworks de este campo: Meteor. Es por ello que esa ha sido la plataforma elegida para este TFG.

La aplicación que se va a llevar a cabo es, concretamente, una aplicación para que el profesor haga un seguimiento del aprendizaje de sus alumnos mediante pequeños cuestionarios que puede realizar todos los días, obteniendo estadísticas en tiempo real de las respuestas que han dado sus alumnos a las preguntas planteadas, pudiendo así saber si los alumnos progresan adecuadamente en la clase. La aplicación, que podrá ser usada en las plataformas móviles más utilizadas actualmente, se centra en ser lo más usable y simple posible, para facilitar tanto al alumno como al profesor la integración de este método en el día a día sin que sea una carga adicional.

La aplicación tiene como fin evaluar presencialmente a los alumnos, permitiendo a los profesores crear cuestionarios rápidamente, que pueden ser publicados directamente y respondidos inmediatamente por los alumnos. Durante la realización del cuestionario, el profesor podrá visualizar de manera sencilla el progreso de sus alumnos, e indagar en los temas en los que los alumnos han cometido más fallos. El TFG pretende también demostrar que, usando esta aplicación, los resultados obtenidos por los alumnos a lo largo de toda su etapa educativa podrían ser mucho mejores si se llegara a usar esta aplicación de manera correcta.

En este documento se explicará más en detalle el desarrollo de esta aplicación de evaluación presencial, detallando lo que pueden hacer con ella alumnos y profesores, y las posibles maneras de emplearla en el actual sistema educativo universitario, así como su arquitectura y estructura, y las explicaciones de las decisiones de diseño tomadas a lo largo de todo el proceso. Además, se explicarán diversos elementos de la plataforma Meteor, qué incluye y la manera de usarla.

Abstract (English)

This Bachelor Thesis is developed with the purpose of investigating the development of web applications with one of the modern platforms that join the latest frameworks and technologies concerning this field: Meteor.

The application made is, more specifically, an app for the teacher to follow the students' learning process by creating questionnaires that can be answered immediately in class every day if necessary. While the students are answering the questions, the teacher can view graphics and information about each question generally or about each student individually. This way, when the students have finished, the teacher has a general idea about their knowledge of the day's explained lesson and can later adapt remaining classes to focus on those questions that were failed. The application will be available in all mobile platforms and its main focus is usability and simplicity, so it's not a burden to use it every day.

The goal of the application is to do a presential evaluation, so the teacher knows which students have attended every class. We also aim to explain how this app could potentially improve student performance if used correctly.

In this document we will explain with detail the Meteor platform, what does it include and how to use it, as well as the specific development of the application explained above, its architecture and design choices made, detailing what teachers and students can do with it and the possible ways of implementing it in the actual university education system.

Palabras clave (castellano)

Meteor, Web, Node, JS, JavaScript, Cuestionarios, Profesor, Alumno, Evaluación Presencial, Tiempo Real, MongoDB, CSS, HTML

Keywords (inglés)

Meteor, Web, Node, JS, JavaScript, Questionary, Teacher, Student, Presential Evaluation, Real Time Application, MongoDB, CSS, HTML

Agradecimientos

A mi familia, que me ha apoyado siempre, a mis amigos, que siempre han estado ahí, y a los profesores que me han enseñado a aprender.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

1.1 Motivación

Este TFG tiene una motivación doble. Por un lado, el desarrollo en sí de una aplicación web específica para la formación presencial de los alumnos. Por otro lado, descubrir y profundizar en el desarrollo de aplicaciones web mediante la plataforma Meteor

Meteor es una de las plataformas que juntan nuevas tecnologías de desarrollo web como node.js, mongodb o angularjs para facilitar el desarrollo de aplicaciones multiplataforma en tiempo real. Dada la gran demanda de dichas aplicaciones actualmente, es interesante investigar la forma en la que se hacen.

La aplicación desarrollada en este TFG consiste en una aplicación para la formación presencial basada en la realización de cuestionarios de forma diaria. La motivación es la mejora en el sistema educativo universitario, ya que el correcto uso de esta aplicación permitiría al profesor saber si los alumnos están asistiendo a las clases que imparte a la vez que observa el nivel de aprendizaje, lo que ayudaría en gran medida tanto a profesor como a alumnos, pudiendo indagarse en las clases en los temas menos comprendidos.

Si se usa correctamente, la aplicación desarrollada podría suponer enormes beneficios tanto para profesores como para alumnos, teniendo un impacto mínimo de esfuerzo para ambos.

1.2 Objetivos

Como se ha especificado en el apartado anterior, este TFG tiene dos objetivos principales. Uno es el desarrollo de una aplicación para la formación presencial. El otro, observar el proceso de desarrollo de una aplicación móvil empleando Meteor

Como sabemos, el método más empleado para la formación presencial es el clásico de "pasar lista", un método tedioso y poco fiable que gasta tiempo tanto de profesores como de alumnos y que únicamente aporta la información de la asistencia de los alumnos, no si atienden en clase, si comprenden las explicaciones del profesor, o su progreso. Esta aplicación se crea con el objetivo de cambiar ese método, ya que gracias a ella se puede conseguir de manera sencilla la asistencia de los alumnos, su progreso durante el curso y su comprensión de la materia impartida, lo que permitiría al profesor adecuar las clases para que sus alumnos reforzasen los temas menos comprendidos.

1.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- **Introducción:** En esta primera parte de la memoria se explican la motivación del trabajo y los objetivos principales que persigue.
- **Estado del arte:** En esta segunda parte de la memoria se explican otros métodos empleados actualmente para conseguir objetivos similares a los de la aplicación desarrollada para este TFG.

- **Diseño:** En esta sección se detallan los requisitos funcionales y no funcionales de la aplicación, así como la arquitectura de la misma y el diseño general, explicando las diferentes partes que la componen y por qué se ha decidido hacerlo de esa manera.
- **Desarrollo:** En esta parte de la memoria se pasa a explicar de manera concisa todos los archivos que forman las partes de las que se habla en el apartado de diseño, las dificultades encontradas y cómo se han solventado para que todo funcione correctamente.
- **Integración y pruebas:** En esta sección se trata el tema de la integración de la aplicación: cómo pasamos del entorno de desarrollo al entorno de producción, donde la aplicación será usada. Además, se aborda el tema de las pruebas que se han realizado sobre ella y el resultado de las mismas.
- **Conclusiones y trabajo futuro:** En esta última sección del cuestionario se muestran las conclusiones extraídas tras la realización del trabajo, qué ha aportado y cuál es el resultado final. Para concluir, se dan ideas de mejoras que pueden llevarse a cabo sobre la aplicación en el futuro.

2 Estado del arte

2.1 Otros métodos

A la hora de llevar a cabo esta aplicación, y tras haber vivido una vida escolar y universitaria, tan solo hay dos métodos comparables a la aplicación desarrollada en este TFG, aunque no son exactamente iguales, comparten varias de las funcionalidades. Estos dos métodos son los siguientes:

2.1.1 Pasar lista

Puede sonar arcaico, pero es el método más empleado para asegurarse de que los alumnos asistan a las clases en la universidad. Se pasa un papel con los nombres de los alumnos de la clase, y en los huecos en blanco el alumno firma, pone una "X"… Este método es tedioso tanto para el profesor como para el alumno, y proporciona una información mínima: simplemente si el alumno ha asistido a clase o no. No permite saber si los alumnos han atendido, han absorbido la lección explicada, etc.

2.1.2 Cuestionarios en plataformas online

Este método se emplea poco, y normalmente con el objetivo de evaluar al alumno (es decir, realizar exámenes) contadas veces. El ejemplo más cercano es Moodle, plataforma que sirve para muchas más cosas aparte de realizar cuestionarios, cosas que esta aplicación no trata de abarcar. Esta aplicación sirve para realizar cuestionarios cortos que sirvan a la vez para pasar lista y evaluar los conocimientos de los alumnos, de manera sencilla tanto para profesores como alumnos.

Figura 2-1: Logo de la plataforma Moodle

3 Diseño

El diseño de esta aplicación se separa en dos partes fundamentales: El modo estudiante y el modo profesor. Ambos, como sus nombres indican, son el acceso a la aplicación por parte de estudiantes y profesores respectivamente. Ambos modos comparten ciertas características, pero también tienen diferencias esenciales.

3.1 Requisitos

3.1.1 Funcionales

- Permitir registrarse en la aplicación con un email y contraseña: Si el email introducido es válido (explicado en el apartado 4.3) se debe registrar al usuario y mandar un correo de verificación al mismo.
- Permitir hacer login en la aplicación con un email y una contraseña: Una vez haya verificado su dirección de email, el usuario podrá acceder a la aplicación mediante dicho email y la contraseña proporcionada.
- Permitir el cambio de contraseña de la aplicación: La aplicación deberá permitir al usuario cambiar su contraseña mediante un link enviado a la dirección de correo del usuario.
- Permitir a los profesores la creación de asignaturas: Los usuarios con el rol de profesor deberán poder crear asignaturas en las que publicar cuestionarios. Una vez creada, el profesor tendrá la opción de generar una clave aleatoria con caducidad para que los alumnos puedan matricularse en ella con dicha clave.
- Permitir a los profesores la gestión de sus asignaturas: Los usuarios con el rol de profesor deberán poder ver un listado con sus asignaturas, para ver estadísticas de las mismas, los alumnos matriculados, etc. y poder cerrarlas cuando crean oportuno.
- Permitir a los profesores la creación y publicación de cuestionarios: Los usuarios con el rol de profesor deberán poder crear cuestionarios para su posterior publicación en las diferentes asignaturas.
- Permitir a los profesores reutilizar cuestionarios pasados: A la hora de crear un cuestionario, se deberá dar la opción a los usuarios con el rol de profesor de elegir entre una "pool" de cuestionarios anteriores, partiendo así de una copia modificable.
- Permitir a los profesores la visualización en tiempo real de estadísticas sobre los cuestionarios en curso: Los usuarios con el rol de profesor podrán ver estadísticas (explicado en el apartado 3.3.2) en tiempo real sobre los cuestionarios publicados y finalizados.
- Permitir a los alumnos matricularse en las diferentes asignaturas disponibles: Los usuarios con el rol de alumno deberán poder matricularse en las asignaturas disponibles, haciendo uso de la clave correspondiente que el profesor deberá dar a sus alumnos.
- Permitir a los alumnos obtener un listado de los cuestionarios abiertos disponibles: Los usuarios con el rol de alumno deberán poder obtener un listado con los cuestionarios abiertos de todas las asignaturas en las que estén matriculados.
- Permitir a los alumnos responder a los cuestionarios abiertos disponibles: Al seleccionar un cuestionario de la lista del requisito anterior, el alumno podrá responder al cuestionario, teniendo una única oportunidad.

- Permitir al alumno ver los cuestionarios realizados anteriormente: Los usuarios con el rol de alumno podrán ver un listado de los cuestionarios que ya han realizado, la fecha y nota de los mismos y todas las respuestas que marcaron a la hora de hacerlo.

3.1.2 No Funcionales

- Aplicación móvil multiplataforma: La aplicación desarrollada debe ser compatible con las plataformas móviles más empleadas en la actualidad.
- Sencillez y facilidad de uso: La aplicación desarrollada debe ser sencilla y fácil de usar, para su fácil integración en el sistema académico.
- Interfaz gráfica: La aplicación debe presentar una interfaz visual que de aspecto de aplicación de móvil nativa.

3.2 Arquitectura

Para llevar a cabo la aplicación, se ha empleado la plataforma Meteor. Esto quiere decir que es una aplicación node.js, en la cual tanto el servidor como el cliente están desarrollados con JavaScript. En el cliente se usarán también plantillas html, combinadas con un lenguaje de plantillas de Blaze llamado Spacebars, que permite crear estructuras como bucles, if, etc. que facilitan mucho la creación de vistas. La base de datos será MongoDB, una base de datos NoSQL que guarda los documentos en JSON (más específicamente BSON), lo que hace que su integración con aplicaciones JS sea más fácil y rápida que una base de datos SQL. Este es un diagrama de la arquitectura de Meteor:

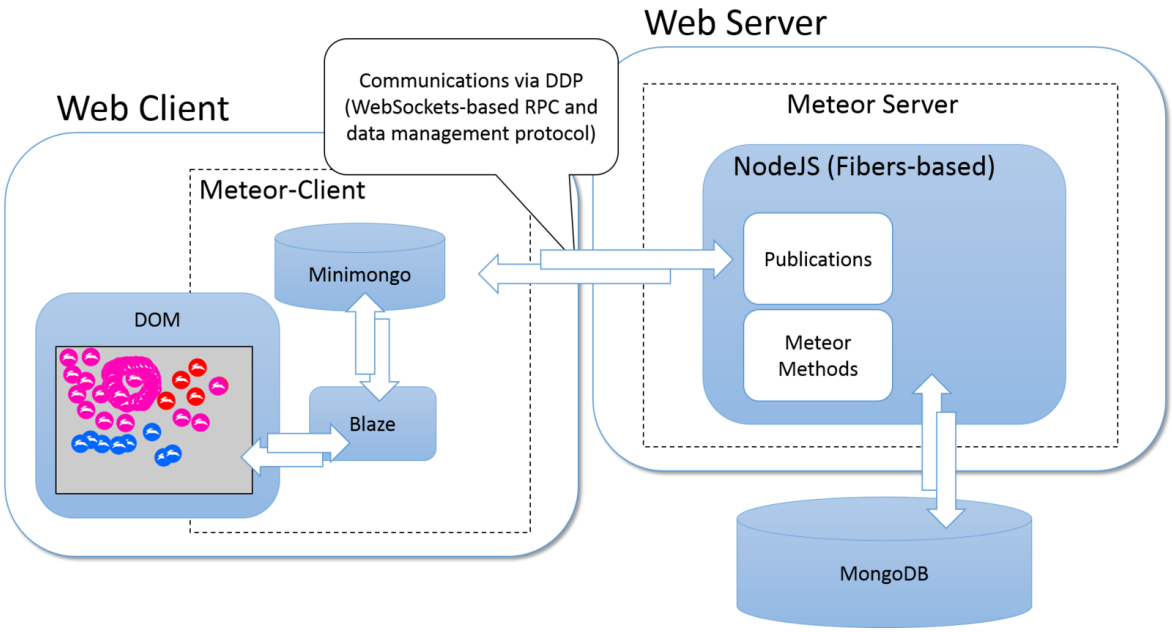

Figura 3-1: Arquitectura de una aplicación Meteor

Como se puede observar en la imagen, la arquitectura es la de cliente-servidor.
La parte del servidor es la única que se comunica directamente con la base de datos MongoDB. En el servidor encontramos **métodos**, que son funciones que se pueden llamar desde el cliente mediante el comando Meteor.call(); y **publicaciones**, que son una parte

muy importante de Meteor: lo que hacen estas publicaciones es especificar qué datos son enviados al cliente cuando este acceda a su propia colección MiniMongo. Por ejemplo, en el caso de la aplicación desarrollada para este TFG, no sería correcto permitir a un alumno obtener las contraseñas o respuestas correctas de un cuestionario cuando se pide la lista de cuestionarios, pero sí sería necesario enviar todas las preguntas de los mismos. En los métodos de publicación se puede escribir código JS como si de un método se tratase, pudiendo así controlar perfectamente qué datos se envían al cliente cuando éste realice la suscripción.
Por otro lado, la parte del cliente no se comunica directamente con la base de datos, sino que lo hace con su propia colección MiniMongo; es una réplica en caché de la base de datos de MongoDB generada a partir de las suscripciones actuales del cliente. Cuando un elemento de la suscripción cambia en la base de datos, se notifica al cliente para que actualice las vistas, lo que facilita el aspecto de tiempo real en la aplicación. En el caso de este TFG se usará Blaze para las plantillas html, lo que permite el uso de Spacebars, un lenguaje de plantillas que permitirá el uso de helpers y otras estructuras para mejorar el aspecto reactivo de la aplicación y facilitar la codificación de las vistas.

3.3 Diseño general

Como ya se ha explicado en el apartado anterior, esta aplicación está desarrollada en Meteor por lo que sigue la arquitectura cliente-servidor con una base de datos en el lado del servidor. La base de datos es MongoDB, que trabaja con colecciones de objetos, el equivalente, podría decirse, a las tablas de una base de datos SQL. No es exactamente lo mismo, ya que una colección MongoDB no impone ninguna restricción sobre cómo han de ser los documentos incluidos en ella. Las colecciones principales de las que hace uso la aplicación son las siguientes:

- **Cursos:** En esta colección se guardan las asignaturas de la aplicación (en un principio se llamaban cursos, de ahí el nombre). Contiene información sobre todas las asignaturas, como los alumnos que tiene cada una, la fecha de creación… Esta información solo es accesible por los profesores, y solo pueden acceder a la información referente a asignaturas que hayan creado ellos mismos.
- **Cuestionarios_profesor:** En esta colección se guardan los cuestionarios creados por los profesores: preguntas, respuestas, respuestas correctas, fecha de publicación… Igual que los cursos, solo pueden acceder a ellos los profesores correspondientes, aunque en este caso, los alumnos matriculados en las asignaturas correspondientes a los cuestionarios podrán ver una versión reducida de los mismos, en la que no se indicará la/s respuesta/s correcta/s, la contraseña (si la hubiera) del cuestionario, etc.
- **Cuestionarios_alumno:** En esta colección se guardan las respuestas de cada alumno a cada cuestionario, con datos adicionales como el tiempo que han tardado en hacer cada pregunta. Los profesores de una asignatura tienen acceso a estos objetos de los alumnos matriculados en ella, y cada alumno tiene acceso únicamente a sus propios cuestionarios.
- **Lista_profesores:** En esta colección se almacenan los correos electrónicos de todos los profesores que accederán a la aplicación.
- **Lista_alumno:** En esta colección se almacenan los sufijos de email que serán interpretados por la app como correos de alumno.
- **Images:** Esta es una colección especial, ya que lo que almacena son imágenes. En la aplicación se usa para almacenar las imágenes de perfil de los usuarios, pero

como se explica en 6.2, podría usarse para más cosas en el futuro, como poner imágenes en las preguntas.

Una vez explicada la base de datos, podemos explicar el funcionamiento de la aplicación. Como se puede observar, una parte básica de este sistema es el acceso a los datos. Es decir, sería crítico que un alumno pudiera obtener un cuestionario al completo, ya que con conocimientos de JS podría obtener las respuestas del cuestionario antes de responder. Por ello es esencial no permitir el acceso a dichos datos, lo que se consigue en Meteor gracias al llamado esquema de **publicación-suscripción:**

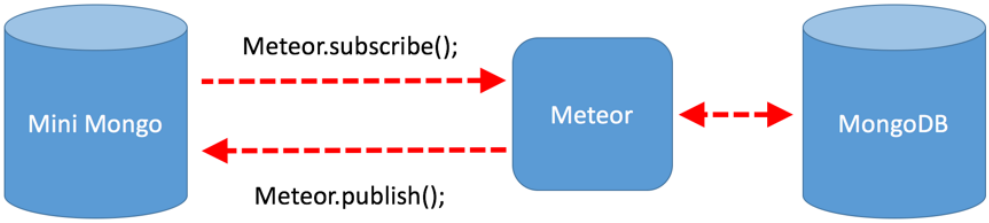

Figura 3-2: Esquema publicación-suscripción

Esto se basa en que únicamente el servidor contacta con la base de datos, y se crean unos métodos que **publican** los datos dependiendo de ciertos parámetros. Luego, el cliente se suscribe a las publicaciones del servidor, y obtiene los datos que éste envía en esa publicación, almacenándolos en su propia colección MiniMongo, lo que permite emplear consultas de MongoDB desde el cliente, sin cambiar en ningún momento la base de datos real. Esto da mucha flexibilidad al desarrollo, además de claridad y limpieza. Aquí podemos ver un ejemplo de una publicación en el servidor:

```
Meteor.publish('lista-cuestionarios', function(idProfesor) {
    //Es un profesor
    if (Roles.userIsInRole(this.userId, ['profesor'], 'default')) {
        return CuestionariosProfesor.find({idProfesor: idProfesor});
    }
    //Es un alumno
    else {
        //Primero hay que ver los cursos en los que se ha matriculado el alumno
        var cursosq = Cursos.find({alumnos: this.userId}, {_id: 1})
        var cursos = cursosq.map(doc => { return doc._id });
        return CuestionariosProfesor.find({curso: {$in: cursos}}, {titulo: 1, _id: 1, curso: 1, dur
    }
});
```

Figura 3-3: Ejemplo de publicación

Dado que es un método de servidor, tenemos acceso a las variables de sesión, por lo que se puede emplear el id del usuario que llama al método para detectar su rol (profesor/alumno) y servir los cuestionarios pertinentes. Además, si el usuario tiene el rol de alumno, se limitan los campos enviados, para no enviar información crítica como las respuestas correctas de cada pregunta. Luego, en el cliente tan solo hay que escribir:

```
self.subscribe('lista-cuestionarios', Meteor.userId());
```

Figura 3-4: Ejemplo de suscripción

Y ya se tendría acceso a los cuestionarios pertinentes, en la colección de MiniMongo que quedará almacenada en la caché.

3.3.1 Casos de uso

En el siguiente apartado explicaremos con detalles los casos de uso del modo profesor y del modo alumno, aunque ahora presentamos un diagrama general de casos de uso:

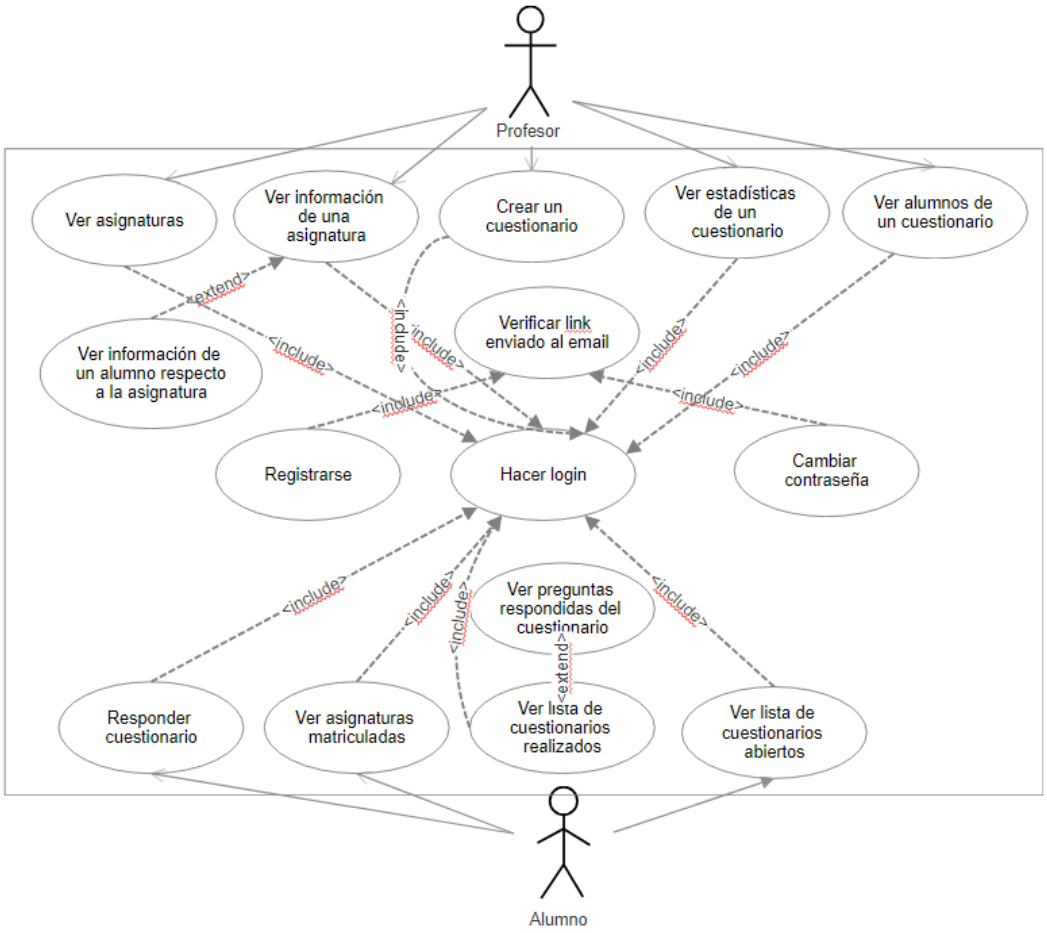

Figura 3-5: Diagrama de casos de uso

Llegados a este punto, hay que separar el diseño de la aplicación en dos modos diferentes: **el modo profesor y el modo alumno.**

3.3.2 Modo profesor

Este es el modo de la aplicación cuando se identifica un profesor en la misma. Para que un usuario sea reconocido como profesor, a la hora de hacer el registro en la aplicación deberá proveer un correo electrónico que se encuentre en la colección **lista_profesores** mencionada anteriormente. Una vez lleve a cabo el registro y verifique su dirección de correo electrónico, podrá acceder a la aplicación con el rol de profesor.

La aplicación tiene varios aspectos principales, ya que un profesor tiene que poder ver listas de cuestionarios y asignaturas, y dentro de cada cuestionario/asignatura tiene que poder ver un listado de alumnos y estadísticas que le permitirán enfocar las clases teniendo

en cuenta los resultados. Desde el menú principal de los profesores se puede acceder a los siguientes elementos principales:

3.3.2.1 Ver cuestionarios abiertos

Esta es una de las partes más importantes de la aplicación. Desde aquí, el profesor podrá ver todos los cuestionarios de sus asignaturas que están abiertos en este momento, es decir, que están siendo realizados. Una vez seleccione uno de los cuestionarios del listado, tendrá varias opciones:

- **Ver alumnos en el cuestionario:** El profesor podrá ver el listado de alumnos que están realizando el cuestionario.
- **Ver alumnos que han finalizado el cuestionario:** El profesor podrá ver el listado de alumnos que ya han terminado el cuestionario.
- **Ver alumnos que no han comenzado el cuestionario:** El profesor podrá ver el listado de alumnos matriculados en la asignatura correspondiente al cuestionario que ni siquiera han entrado al cuestionario.
- **Ver estadísticas globales:** El profesor podrá ver una serie de estadísticas en tiempo real sobre el cuestionario: Un histograma con los alumnos que han respondido correctamente a cada pregunta, la nota media y el tiempo medio del cuestionario, la pregunta más y menos acertada y la pregunta que más y menos han tardado los alumnos en contestar. Estas estadísticas pueden ser útiles para que el profesor sepa qué temas resultan más complicados a los alumnos.
- **Ver preguntas:** El profesor podrá ver estadísticas independientes por pregunta, presentándose una gráfica con los alumnos que han contestado correcta o incorrectamente o no han contestado a cada una de las cuestiones, así como el tiempo medio que han tardado los alumnos en contestar.

3.3.2.2 Administrar mis asignaturas

Desde esta sección el profesor puede ver un listado con todas las asignaturas que ha creado, filtrando por múltiples parámetros como el nombre, fecha o si esa asignatura está cerrada o no. Desde este listado podrá también añadir nuevas asignaturas. Una vez seleccionada una asignatura del listado, tendrá acceso a varias opciones:

- **Ver alumnos matriculados:** El profesor podrá ver el listado de alumnos matriculados en la asignatura, apareciendo en dicho listado la nota media de cada alumno en esa asignatura. Podrá filtrar por el nombre y nota de los alumnos. Si selecciona un alumno de la lista, podrá ver gráficas con la asistencia del alumno y su nota a lo largo del curso, comparada con la asistencia y nota promedio del resto de los alumnos de la asignatura.
- **Ver estadísticas de la asignatura:** El profesor podrá ver dos gráficas de los alumnos de la asignatura como conjunto: la asistencia a clase y la nota promedio.
- **Ver cuestionarios:** El profesor podrá ver un listado con todos los cuestionarios de la asignatura, ya sean realizados anteriormente o publicados actualmente. En cada cuestionario podrá ver las estadísticas mencionadas anteriormente, los alumnos que lo hicieron, etc.
- **Generar clave de acceso a la asignatura:** Para que los alumnos puedan matricularse en una asignatura, el profesor debe abrir el periodo de matriculación de dicha asignatura. Para ello, ha de seleccionar el tiempo que quiere que esté abierto dicho periodo, lo que generará una clave aleatoria que caducará en el tiempo especificado, con la cual los alumnos podrán matricularse en la asignatura.

En el caso de que hubiera una clave ya generada, en este apartado se verá dicha clave.

- **Cerrar asignatura:** Normalmente, las asignaturas en la universidad suelen durar un semestre, pero pueden darse casos de que duren más. Tras pulsar este botón y confirmar la acción, la asignatura en cuestión quedará cerrada y archivada. Esto significa que los alumnos matriculados en ella ya no recibirán notificaciones de ningún tipo, y el profesor no podrá realizar ninguna publicación en ella. Tanto profesores como alumnos podrán ver todo lo que podían ver en una asignatura abierta, pero notificándose que es una asignatura de años anteriores.

3.3.2.3 Administrar borradores

En este apartado el profesor podrá ver un listado de todos los cuestionarios que haya guardado como borradores. Una vez seleccione un cuestionario de la lista se le presentarán dos opciones.:

- **Editar cuestionario:** Se presentará el mismo proceso que para crear un cuestionario, solo que estarán rellenos los campos que el profesor rellenó cuando guardó el cuestionario como borrador. Las preguntas y respuestas también quedarán guardadas, pudiendo así añadirse o borrarse preguntas/respuestas.
- **Publicar cuestionario:** Se publicará el cuestionario seleccionado. El proceso de cierre de cuestionario se planificará para la hora definida por el profesor (hora actual + duración del cuestionario) y ya no podrá editarse más.

3.3.2.4 Añadir nuevo cuestionario

En este apartado, el profesor podrá crear un nuevo cuestionario (debe tener alguna asignatura creada) paso a paso o seleccionar un cuestionario realizado con anterioridad para usarlo como plantilla. El principal objetivo de este apartado, a parte de la funcionalidad, es la simplicidad: el profesor debe ser capaz de añadir un nuevo cuestionario rápidamente, sin tener que buscar las opciones, es decir, ha de ser intuitivo.
La creación del cuestionario consta de tres partes:

- **Parte 1:** En esta parte, el profesor ha de introducir los datos básicos del cuestionario: título, tema, asignatura, duración y, si lo desea, una contraseña de acceso.
- **Paso 2:** En esta parte se añadirán las preguntas y las respuestas a dichas preguntas. Tras pulsar el signo "+" situado en la parte superior derecha de la tabla de preguntas, se presentarán los campos necesarios para crear una pregunta: el enunciado y las respuestas. Se pueden añadir tantas respuestas como se quiera, y pueden ser correctas varias (opción múltiple). Si se mantiene pulsada una respuesta, aparecerá un menú contextual con múltiples opciones: marcar una respuesta como correcta/incorrecta, borrarla o moverla hacia arriba o hacia abajo en la lista de respuestas. Se pueden añadir tantas preguntas como se desee. En este apartado encontramos también la opción ***orden aleatorio de las respuestas***. Si se marca esta opción, las respuestas aparecerán en un orden aleatorio para cada alumno, en lugar de aparecer todas en el mismo orden para todos.
- **Paso 3:** En este apartado se presentan únicamente dos opciones: Publicar o guardar como borrador. Si se publica el cuestionario, este quedará abierto y no se podrá editar más. Si se guarda como borrador, los alumnos no podrán verlo aún, y podrá ser editado más adelante tal y como se explica en el apartado 3.3.1.3.

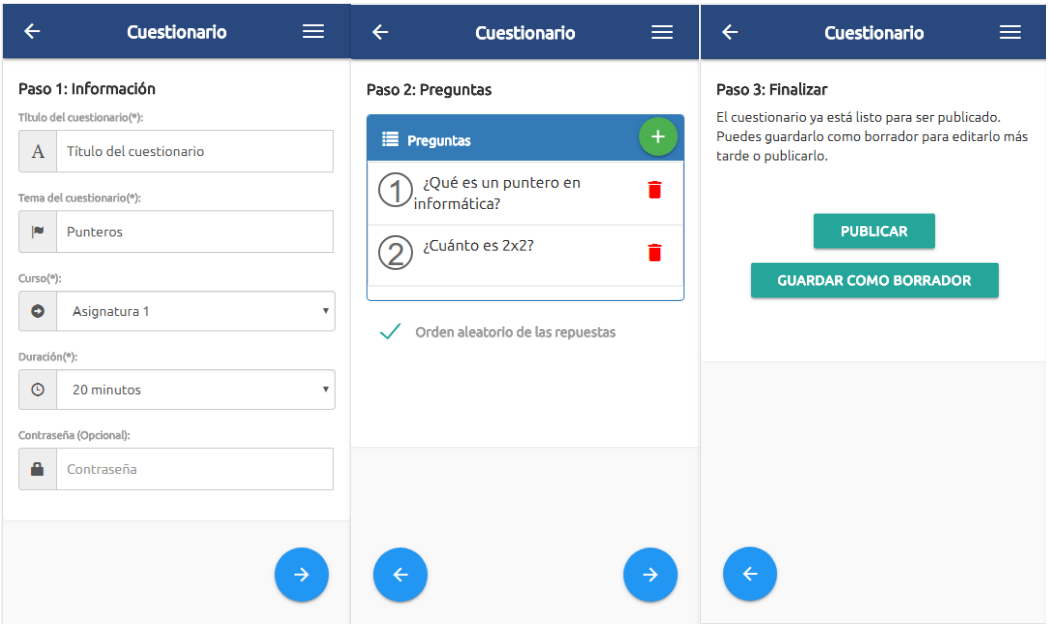

Figura 3-6: Creación del cuestionario en 3 pasos

3.3.2.5 Cambiar contraseña

En este apartado, el profesor podrá seguir los pasos indicados en un email que se enviará a su dirección de correo para cambiar la contraseña de su cuenta.

3.3.3 Modo alumno

Este es el modo de la aplicación cuando se identifica un alumno en la misma. Para que un usuario sea reconocido como alumno, a la hora de hacer el registro en la aplicación deberá proveer un correo electrónico cuyo sufijo (ej. @alumno.com) se encuentre en la colección **lista_alumnos** mencionada anteriormente. Una vez lleve a cabo el registro y verifique su dirección de correo electrónico, podrá acceder a la aplicación con el rol de alumno. El alumno no tiene forma de modificar la base de datos salvo para contestar a un cuestionario. Es decir, no puede publicar cuestionarios. Tampoco puede ver notas de otros alumnos, aunque sí podrá ver las suyas, y los cuestionarios que ha realizado a lo largo del tiempo. Desde el menú principal de los alumnos se puede acceder a los siguientes elementos principales:

3.3.3.1 Matricularse en una asignatura

En este apartado, el alumno podrá ver un listado de las asignaturas disponibles en las que puede matricularse. Una vez seleccione una asignatura de la lista, se le pedirá que introduzca la clave de acceso facilitada por el profesor. Si la clave introducida es correcta, el alumno quedará matriculado en la asignatura y tendrá acceso a los cuestionarios que se publiquen en la misma. En este listado no aparecerán asignaturas en las que el alumno ya esté matriculado.

3.3.3.2 Ver asignaturas matriculadas

En este apartado, el alumno podrá ver un listado con todas las asignaturas en las que se ha matriculado, pudiendo filtrar por diversos parámetros como el nombre o si es una asignatura en curso o una antigua. Si selecciona una asignatura, podrá ver información sobre ella, aunque no podrá hacer nada más. En el apartado 6.2 se explican posibles mejoras futuras relativas a este apartado.

3.3.3.3 Ver cuestionarios realizados

En este apartado, el alumno podrá ver un listado con todos los cuestionarios que ha realizado a lo largo del tiempo, incluyendo aquellos que realizó años pasados. Podrá ver la fecha en la que realizó cada cuestionario, así como la nota obtenida. Si selecciona un cuestionario de la lista, podrá ver lo que respondió a cada pregunta, y las respuestas correctas.

3.3.3.4 Ver cuestionarios abiertos

En este apartado, el alumno podrá ver un listado de los cuestionarios abiertos que estén publicados en sus asignaturas. Si selecciona un cuestionario de la lista, en caso de que este tuviera contraseña, se le pedirá que la introduzca. Si la contraseña introducida es correcta, se dará acceso al alumno al cuestionario. Una vez se inicie el cuestionario, el alumno no podrá salir del mismo, y podrá responder o dejar en blanco las preguntas que crea oportuno. Una vez finalizado el cuestionario, el alumno podrá ver la nota obtenida, pero no podrá repetirlo.

3.3.3.5 Cambiar contraseña

En este apartado, el alumno podrá seguir los pasos indicados en un email que se enviará a su dirección de correo para cambiar la contraseña de su cuenta.

4 Desarrollo

4.1 Fases de desarrollo

Aunque sea una aplicación móvil desarrollada con Meteor, el TFG no deja de ser una aplicación web, por lo que podríamos decir que sigue un patrón modelo-vista-controlador, en el que están claramente separadas la vista y el código de la aplicación. Por ello separar el desarrollo en dos partes: la creación y modificación de las plantillas html y la codificación necesaria para que se comuniquen con el servidor y obtengamos el contenido deseado.

Aunque podamos separar el desarrollo en estas dos partes, normalmente una condiciona a la otra. Es decir, en este caso, para cada funcionalidad deseada, el primer paso es **crear la plantilla html.** Una vez creada la plantilla html con los elementos variables definidos, se crea **el código del cliente.** Este código es esencial, ya que es el que se encarga de llamar a las funciones (en el caso de Meteor, se llaman **métodos**) del servidor y cambiar la vista con los datos obtenidos. Como se explica en el apartado 5.2, el hacer un método de pruebas de caja negra favorece este tipo de desarrollo, ya que podemos crear la función en el cliente sabiendo que, una vez esté hecha en el servidor, devolverá los datos correctos, de los que sabremos el tipo, tamaño, etc. con antelación. Por último, se creará **el código del servidor**. Este código es esencial, ya que el servidor es el único que puede comunicarse con la base de datos.

Sabiendo que son tres partes separadas pero muy unidas a la vez, y que no pueden realizarse unas sin pensar en el resto, vamos a pasar a explicar la forma de desarrollo de cada una de las partes descritas.

4.1.1 Creación de las plantillas

Por muchos es considerada la parte más sencilla de una aplicación web, pero en mi opinión, la creación de las plantillas es tan importante como el resto e igual sino más difícil de llevar a cabo. La mayoría de las plantillas han cambiado ligeramente a lo largo de todo el desarrollo, ya que la idea inicial de la plantilla no siempre se mantiene. Además, en Meteor, las plantillas tienen un lenguaje propio: **Spacebars**. Este lenguaje nos permite hacer bucles y condicionales que ayudan a crear la plantilla. Pongamos como ejemplo el listado de cuestionarios de la aplicación: es una plantilla html muy sencilla, ya que lo único que tiene (además de la estructura) es un bucle *for* que recorre el objeto *cuestionarios* e imprime uno a uno cada cuestionario, con los datos relevantes. El funcionamiento de este código se explicará más adelante en el apartado 4.1.2.

Dejando a un lado Spacebars, la creación de estas plantillas no difiere de la creación de plantillas html tradicionales. En estas se ha puesto especial empeño en que parezcan una aplicación nativa de móvil, para que al usuario le resulte más cómodo hacer uso de la aplicación. Aunque sea una de las partes que más se tarda a la hora de desarrollar, es una de las que menos hay que hablar, ya que podríamos decir que una plantilla es correcta cuando al usuario le parece cómoda e intuitiva. Para cargar las plantillas, Meteor dispone de varios métodos, pero he optado por usar **Blaze,** que es el que viene de fábrica en las últimas versiones de Meteor. Como se explicó anteriormente, la estructura de las vistas de la aplicación no cambia, manteniéndose el estilo de cabecera-cuerpo a lo largo de todas las vistas.

Acompañando a los archivos html se encontrarán las hojas de estilos css, en las que se especifican los estilos de diversas clases e ids, y haciendo uso de CSS3 conseguimos un

resultado visual sencillo y simple a la vez que cuidado, lo cual lleva una gran cantidad de tiempo.

4.1.2 Creación del código del cliente

Desde el principio de la memoria se ha hablado de la arquitectura de una aplicación en Meteor, más específicamente de cómo tan solo el servidor puede comunicarse directamente con la base de datos. Pero para que el usuario pueda visualizar dichos datos es necesario tenerlos en el cliente, y ahí es donde entra en juego el código que vamos a explicar a continuación.

Como bien sabemos, con JavaScript en el cliente se pueden hacer multitud de cosas, como manipular el DOM o enviar peticiones HTTP. En la aplicación desarrollada para el TFG se usa para ambas cosas, en conjunción de JQuery. No es necesario hablar de cómo se emplean estas tecnologías para manipular el DOM, ya que no se hace de manera distinta a lo habitual, pero sí para comunicarse con el servidor, ya que esto se hace de una manera diferente: se usa el comando *Meteor.call()* para llamar a los métodos Meteor definidos en el servidor (explicado en el apartado 4.1.3). Estas llamadas son **asíncronas**, es decir, no paran la ejecución del código cuando se hacen. Como casi cualquier función asíncrona que nos podamos encontrar, permiten pasarle como argumento una función de callback que se ejecutará una vez el método del servidor haya hecho el return. Como se explicará en el siguiente apartado, un método de Meteor en el servidor puede devolver o bien un objeto o un error. La función de callback mencionada anteriormente tiene como parámetros estos dos elementos, pudiendo comprobarse si ha devuelto error y en caso afirmativo de qué error se trata, permitiendo llevar un buen control de estos.

La forma de navegar entre vistas es mediante el enrutador ***FlowRouter***, que se explica con más detalle en el apartado 4.2.2. Por ello, cuando queramos cambiar la vista en la que nos encontramos basta con llamar a la función *FlowRouter.go()* con los parámetros pertinentes.

Hasta el momento hemos hablado de código JavaScript, pero como comentábamos en el apartado anterior, no todo el código del cliente está escrito en este lenguaje. Hay código escrito en el lenguaje de plantillas **Spacebars**. Este código nos permite crear estructuras que facilitan la creación de plantillas, ya que nos permite recorrer y comprobar los objetos pasados como argumentos a la plantilla, el rol de un usuario y más cosas. Es una característica extremadamente útil, y que favorece la reutilización de plantillas evitando así la repetición innecesaria de código. Si se quiere, por ejemplo, mostrar un listado de objetos (que se extraen de distintos lugares dependiendo del contexto) basta con renderizar la misma plantilla, pero con listas de objetos diferentes (siempre y cuando sean objetos con los mismos atributos) y obtendríamos el mismo resultado que creando varias plantillas, una para cada caso. Como ejemplo claro en la aplicación, tenemos la plantilla que muestra el listado de asignaturas, que son diferentes dependiendo de si se quieren ver las de un alumno, las de un profesor, o las que están abiertas para que se puedan matricular en ellas. Sin Spacebars, habría que buscar la manera de automatizar esto, ya que es una plantilla dinámica, que depende del número de asignaturas a mostrar y de los datos de las mismas. Además, en el caso de que no hubiera asignaturas disponibles (el alumno ya se ha matriculado de todas, están ocultas porque el profesor no ha abierto el periodo de matriculación…) debería mostrarse un mensaje que explicase que no hay asignaturas disponibles. Sin Spacebars, habría que hacer código JS que obtuviera los datos de las asignaturas y los incrustase en el DOM. A continuación, podemos ver un extracto de código de dicha plantilla usando Spacebars, y el resultado html que produce:

```
<div class="profile-usermenu" id="lista-cursos">
    <ul class="nav">
        {{#if $eq cursos.count 0}}
            <li>
                <a><div>No hay cursos</div></a>
            </li>
        {{else}}
            {{#each cursos}}
            <li class="curso-lista" cursoid={{_id}}>
                <a cursoid={{_id}}>
                    <i class="glyphicon glyphicon-book"></i>
                    <div style="display:inline;" cursoid={{_id}}>{{nombre}}</div>
                    {{#if tienePassword}}
                        <i class="glyphicon glyphicon-lock" style="float:right;"></i>
                    {{/if}}
                </a>
            </li>
            {{/each}}
        {{/if}}
    </ul>
```

Figura 4-1: Código Spacebars

Figura 4-2: Resultado del código Spacebars

Además de todo esto, hay un elemento más acerca de las plantillas desde el punto de vista de codificación que es esencial: los estados. En Meteor, las plantillas pasan por tres estados principales que podemos capturar para ejecutar cierto código en el momento oportuno:

- **OnCreated:** Esta función se llama cuando se ha creado la plantilla, pero aún no ha sido renderizado. Es el lugar adecuado para realizar las suscripciones necesarias para la vista que se acaba de crear.
- **OnRendered:** Esta función se llama cuando la plantilla ha terminado de renderizarse. Es el momento idóneo para realizar cambios sobre el DOM dependiendo de otros parámetros, como por ejemplo cambiar el texto de la navbar superior de la aplicación para que coincida con el de la página en la que nos encontramos.
- **OnDestroyed:** Esta función se llama una vez se ha destruido la plantilla. Es útil para enviar información al servidor o para terminar acciones que únicamente deberían pasar en la vista en la que nos encontrábamos.

A parte de estos estados, las plantillas tienen los llamados eventos, que son exactamente los mismos que en JavaScript: click, hover, focus… Es decir, eventos que hacen trigger cuando ocurre una acción determinada en un elemento del DOM, que pueden ser capturados para ejecutar código específico cuando sucedan. La forma que tiene Meteor de definir este código para los eventos es mediante un objeto en JSON pasado al objeto *events*, atributo de cada plantilla. En dicho JSON se especificará el elemento en el que ocurre la acción, la acción que ha de ocurrir y el código que ha de ejecutarse cuando ocurra, por ejemplo:

Por último, es necesario hablar de los métodos **helper**, que son esenciales para hacer la aplicación en tiempo real y que de la sensación al usuario de que se trata de una aplicación nativa y no de una página web que haya que recargar tras cada petición, que es uno de los objetivos de este TFG. Estos métodos se definen en el cliente, en la parte del código JavScript llamada helpers, que es un atributo de las plantillas, al igual que los eventos y estados de los que hablamos anteriormente. Los helpers son pequeñas funciones con un valor de retorno, que pueden ser llamadas desde la plantilla mediante Spacebars, imprimiéndose el resultado en pantalla como parte del html generado. Además, dado que en Meteor las colecciones son reactivas, si un helper emplea una colección, cuando dicha colección sufre un cambio en la base de datos, el método vuelve a ejecutarse con la colección actualizada, lo cual permite que muchas partes de la aplicación operen en tiempo real.

4.1.2.1 Plugin Chartist

Como hemos explicado anteriormente, un elemento fundamental de la aplicación son las gráficas, mediante las cuales, el profesor podrá ver rápidamente el resultado de un cuestionario, o el progreso de un alumno en una asignatura a lo largo del tiempo. Pero crear el código que dibuje los diferentes tipos de gráficas a partir de datos es una tarea muy compleja que requiere conocimientos gráficos, por ello se ha decidido emplear un plugin existente llamado Chartist. Chartist nos permite crear varios tipos de gráficas (histogramas, gráficas de puntos, gráficos circulares…) pasando los datos correctamente formateados a las funciones correspondientes de este plugin. Es por eso que, gracias a este plugin, la dificultad de hacer las gráficas de la aplicación reside en obtener los datos de la base de datos, generar las estadísticas necesarias y formatearlas correctamente.

4.1.3 Creación del código del servidor

El código creado en el servidor no difiere mucho del código habitual de los servidores a los que estamos acostumbrados, salvo porque también está escrito en JavaScript. La parte del servidor consiste en una serie de métodos que esperan un parámetro data, que es un objeto JSON. Estos métodos tienen acceso a las variables de sesión, como el id del usuario que ha llamado al método, y pueden devolver o bien un error o bien un resultado. Como se comentó anteriormente, la llamada a estos métodos es asíncrona.
Además de esos métodos, en el servidor se encuentra un archivo en el que se instancian las colecciones de la base de datos, para que puedan ser usadas desde JavaScript como si de un objeto se tratase.

4.2 Estructura final de la aplicación

Como hemos dicho desde el principio, como cualquier aplicación web, este TFG tiene principalmente una parte de cliente y una parte de servidor. Al tratarse de un proyecto Meteor, la estructura del proyecto difiere algo de las estructuras que estamos

acostumbrados a ver en otras ocasiones como puede ser un proyecto php. Tras todos los cambios que hemos explicado en el apartado anterior, la estructura final de la aplicación sería la que se ve a continuación.

4.2.1 Client

En esta carpeta, como su nombre indica, van todos los archivos relacionados con el cliente. Los tipos de archivo que encontraremos en esta carpeta son .html, .js y .css. Para organizar más el proyecto, he creado una carpeta por cada view de la aplicación. En cada carpeta se encuentran los archivos relevantes para dichas views.

La forma de cargar los templates para que se vean desde el cliente es la siguiente: hay un template principal, **main.html**, que contiene la estructura de todas las views. Dicha estructura es la de una navbar en la parte superior de la aplicación y el cuerpo de la aplicación propiamente dicho debajo de la navbar. De esta forma, cuando queramos cargar un template, hay que pasarle al template principal los argumentos del template de la navbar y del template del cuerpo. Esto permite tener varias navbars diferentes manteniendo la misma estructura, lo cual tan solo se usa en esta aplicación cuando el usuario aún no ha hecho login, ya que la navbar no contiene el menú de usuario. En la parte main.js se cargan las dependencias necesarias para el correcto funcionamiento de la aplicación (como el enrutado, que se explicará más adelante en el apartado 4.1.2).

4.2.2 Imports

En esta carpeta se guarda el código js que puede ser referenciado desde el resto de los archivos js del proyecto. Este apartado se divide a su vez en dos carpetas: client, en la que se guardarán los imports relacionados con el cliente; y server, en la que se guardarán los imports relacionados con el servidor.

En este caso, la carpeta server estará vacía, pero la carpeta client contiene archivo clave para el desarrollo del proyecto: **routes.js**. Este archivo es el que indica a la aplicación qué hacer cuando el usuario introduce una dirección determinada. Aquí podemos ver un ejemplo de código para direccionar el perfil o página principal de la aplicación:

```
FlowRouter.route( '/profile', {
  name: "profile",
  action: function() {
    BlazeLayout.render('principal', {main: 'profile', navbar:'navbar_menu'});
  }
});
```

Como podemos observar, lo que ocurre aquí es que cuando el usuario introduzca la ruta */profile*, la aplicación renderizará el template principal, pasándole como argumentos el template *profile* para el template principal que comentamos en el apartado anterior y el template *navbar_menu* para el template de la navbar superior. En este archivo también se pueden llevar a cabo más acciones, como por ejemplo que si el usuario no está identificado y trata de acceder a una ruta que no sea login, se le redireccione automáticamente a la página de login.

4.2.3 Lib

En esta carpeta hay un único archivo js: *collections.js*. En este archivo se crean las colecciones que usaremos posteriormente en todo el proyecto Meteor, dando un nombre

para la colección de MongoDB. Es decir, se crea un objeto JS equivalente a una colección mongo, mediante el cual tendremos acceso de manera sencilla a la base de datos en todo momento. Esto se consigue gracias al objeto Mongo.Collection, que como hemos dicho, hace de intermediario entre la base de datos y la aplicación.

4.2.4 Node_modules

En esta carpeta se guardan todos los plugins de Node.js que se hayan añadido a la aplicación. Los principales de esta aplicación son los encargados de dibujar gráficas a partir de datos, una tarea extremadamente compleja de realizar de no ser por el plugin *Chartist.js*.

4.2.5 Packages

En esta carpeta se guardarán todos aquellos paquetes locales que se vayan a usar en el proyecto. En este caso tan solo hay un paquete que no sea de Node.js, y que, por consecuencia va en esta carpeta: *materialize-modal*. Este paquete se descargó directamente de github, y todo el código fuente del mismo se guarda en esta carpeta.

4.2.6 Private

En esta carpeta irán todos los archivos que no sean código que únicamente pueden ser accedidos desde el servidor. En nuestro caso, tan solo están los archivos html que se usarán como plantilla a la hora de enviar emails al usuario que requieran un estilo complejo: cuando se cierra un cuestionario, se envía al profesor que lo creó un email con estadísticas, que se detalla más en el apartado 4.3.

4.2.7 Public

En esta carpeta se guardarán los archivos que no sean código que pueden ser accedidos desde el cliente. En nuestro caso, dichos archivos son las fotos de perfil de los usuarios. Todos los usuarios comienzan con una foto de perfil genérica que pueden cambiar tantas veces quieran pulsando en la imagen del perfil y añadiendo una nueva imagen.

4.2.8 Server

En esta carpeta se guarda todo el código relativo al servidor. Como hemos explicado con anterioridad, este es el único código que tiene acceso a la base de datos. Se han separado los archivos según la funcionalidad de cada uno, para mantener orden en esta parte y facilitar la adición de nuevas características. Más concretamente podemos ver los siguientes archivos JavaScript:

- **Cuestionarios:** En este archivo se encuentran todos los métodos relacionados con la creación y edición de cuestionarios.
- **Cursos:** En este archivo se encuentran los métodos necesarios para crear, abrir y cerrar asignaturas.
- **Db:** En este archivo se encuentran todos los métodos de publicación de la base de datos, y cualquier colección que se añadiese a la base de datos en un futuro requeriría la modificación de este archivo.
- **Email:** En este archivo se encuentra todo el código necesario para poder enviar emails con la aplicación.
- **Estadisticas:** En este archivo se encuentran los métodos encargados de extraer los datos de la base de datos y generar estadísticas formateadas adecuadamente para que el cliente pueda mostrarlas en forma de diferentes gráficas.
- **Main:** En este archivo se hacen los imports necesarios (que se extenderán al resto de archivos del servidor) y se crea el código de startup, es decir, el código que

correrá cuando la aplicación se inicie. El código que va en esta parte es el de el gestor de trabajos, que se explica con más detalle en el apartado 4.2.8.1.

- **Users:** En este archivo se encuentra el código necesario para la creación de usuarios y el cambio de contraseña de estos, así como las opciones del paquete accounts relativas a si se pueden crear usuarios desde el cliente (lo cual se prohíbe, obligando al cliente a enviar la petición de creación de usuario a un método del servidor para que se compruebe si se debe dejar crear la cuenta a dicho usuario) y si un usuario sin verificar (esto es, que no ha entrado en el link enviado por email cuando se creó la cuenta en la aplicación) puede hacer login, lo cual no se permite.

4.2.8.1 Gestor de trabajos: SyncedCron

Durante el desarrollo de la aplicación se requiere una forma de hacer que tanto los cuestionarios como las claves autogeneradas de las asignaturas caduquen. En el caso de los cuestionarios, caducar significaría que los alumnos ya no pueden acceder a ellos para realizarlos, y en el caso de las claves de asignatura significaría que, para que nuevos alumnos puedan matricularse en ella, el profesor de la misma deberá generar una nueva clave. Decimos que estos dos elementos "caducan" porque tienen una fecha de expiración, en la que ocurre esto. Aquí es cuando entra en juego el plugin *SyncedCron*. Este módulo nos permite crear trabajos (esto es, funciones JavaScript) que se ejecuten en una fecha determinada. Dicha fecha puede ser exacta, o puede ser del estilo de todos los lunes a las 12:00, el primer mes de cada año, la segunda semana de cada mes… Aunque en esta aplicación sólo se usará la fecha exacta. Como hemos dicho al principio del apartado, hay dos momentos en los que emplearemos este módulo:

- **Para cerrar un cuestionario cuya duración ha terminado:** Cuando un profesor publica un cuestionario, el servidor planifica un trabajo de cierre del mismo para la fecha especificada por el profesor (fecha en la que se publica + minutos de duración especificados en el cliente). Este trabajo se encarga de poner el estado del cuestionario a **finalizado** en la base de datos y de enviar un email con las estadísticas generales del cuestionario al profesor que lo publicó.
- **Para destruir la clave de acceso a una asignatura:** Cuando un profesor genera una clave de acceso a una asignatura, permitiendo así que los alumnos se matriculen en ella haciendo uso de dicha clave, especifica también el periodo de duración de esta clave, pudiendo elegir entre una hora, un día o una semana. Pasado ese periodo, es necesario destruir la clave y cerrar la matriculación en esa asignatura. Cuando se genera la clave, se crea también un trabajo que, llegada la fecha especificada realiza esta tarea.

Pero es necesario hacer uso de SyncedCron en un lugar más: en la función de startup. En caso de que la aplicación se reiniciase, los trabajos creados en memoria cuando el profesor publica un cuestionario o genera una clave para una asignatura, desaparecerían. Por ello, en la función de startup se comprueba en la base de datos qué trabajos hay que crear, para dejar planificados los pertinentes para cerrar los cuestionarios abiertos y eliminar las claves generadas en curso, asegurando así el correcto funcionamiento de la aplicación.

4.3 Envío de emails

En un principio, la aplicación no hacía uso del módulo de emails de Meteor. Para detectar si un usuario era alumno o profesor, simplemente se miraba el sufijo de la dirección de correo, y si coincidía con el establecido en el centro (que se obtenía de un archivo js en el proyecto) se creaba y autentificaba la cuenta. Como es fácil observar, esto tiene graves consecuencias: si un usuario conoce el email de otro alumno/profesor, puede hacerse una cuenta antes que él, y hacer un uso indebido de la aplicación. Por ello, la solución por la

que se ha optado es la de enviar emails de verificación a la hora de crear una cuenta de usuario. Para ello, es necesario ajustar la variable de entorno **MAIL_URL,** que apunta a una dirección smtp desde la que se pueden enviar correos. Existen múltiples alternativas en el mercado, pero la que se ha escogido para hacer las pruebas en la fase de desarrollo ha sido mailgun, que ofrece un plan de prueba gratuito. La configuración de dicha variable se explica con más detalle en el Anexo A.

Una vez configurado el módulo de email, hay que decirle a la aplicación que no deje logearse a usuarios no verificados. El módulo de usuarios de Meteor, cuando se registra un usuario, además de los campos de email, nombre, contraseña, etc. tiene un campo para la dirección de email que indica si esta ha sido verificada o no. En la parte del servidor, hay que ajustar la función *Accounts.validateLoginAttempt* para que devuelva true solo si el usuario que está haciendo login ha verificado su dirección de correo. Cuando un usuario que no ha verificado su dirección de email trate de hacer login en la aplicación, se comprobará automáticamente esta función, y, en caso de que devuelva false, la función de login devolverá un error.

Tanto cuando se mande el email de verificación como el de cambio de contraseña, se redirigirá al usuario a una página donde se le darán las instrucciones necesarias para llevar a cabo la acción.

Pero además de estos correos, la aplicación hace uso del módulo en un caso más: cuando un cuestionario termina, se envía un correo al profesor de la asignatura con las estadísticas principales y un archivo adjunto JSON con todos los resultados del cuestionario. Este archivo puede carecer de interés para el profesor, pero puede emplearse en el futuro en alguna mejora de la aplicación o módulo externo. Dado que cada correo es diferente, ya que tiene diferentes datos, y la forma de enviar correos es enviando una plantilla html, es necesario renderizar el html en el servidor como si de una plantilla Meteor se tratase, para que las variables de la misma tomen los valores correctos. Esto se puede hacer gracias a un módulo adicional de Meteor, que permite exactamente eso.

5 Integración, pruebas y resultados

5.1 De entorno de desarrollo a entorno de producción

Para escribir este apartado de la memoria es necesario explicar cómo funciona Meteor en cuanto a entornos de producción-desarrollo se refiere. Cuando comenzamos a escribir la aplicación, y comenzamos a probarla, el código corre con el comando *meteor run*. En cambio, cuando queramos publicar la aplicación en un entorno de producción, es decir, para su uso real, se hace de manera diferente. Vamos a explicar ambas maneras y por qué no se debe usar *meteor run* para el entorno de producción:

Comencemos hablando de *meteor run*. Como hemos dicho antes, este comando lanza la aplicación. Lo hace con ciertos parámetros por defecto: en el puerto 3000, con una base de datos por defecto, en localhost… pero con un elemento clave: un detector de cambio de los archivos del proyecto. Esto quiere decir que, cada vez que modifiquemos un archivo, Meteor lo detectará y volverá a correr la aplicación con los nuevos cambios. Esto es extremadamente útil durante el desarrollo de la aplicación, ya que no tenemos que preocuparnos de parar y volver a lanzar el proyecto cada vez que cambiamos algo de código. Además, esto ocasionaba que cada vez que se guardaba una imagen, por ejemplo, cuando el usuario sube una nueva foto de perfil, Meteor hiciera que todos los clientes recargasen la página. Esto puede no parecer demasiado crítico, pero lo es: un alumno haciendo un cuestionario, o un profesor creando uno, y de repente la página se recarga y pierde el progreso. Por último, esta característica tiene otro elemento que afecta negativamente al entorno de producción, y ese elemento es la carga de CPU. Como podemos imaginar, detectar cambios en archivos requiere que la CPU esté constantemente buscando dichos cambios, lo que añade una carga innecesaria en el entorno de producción, donde los archivos no deberían cambiar a no ser que fuera una actualización de la aplicación, en cuyo caso el servidor se cerraría temporalmente hasta que la actualización hubiese finalizado.

Una vez explicado por qué tener la aplicación en entorno de producción con *meteor run* no es una buena idea, vamos a explicar la manera de preparar la aplicación para el entorno de producción. Cabe destacar que Meteor está pensado para el desarrollo de aplicaciones móviles, por lo que permite hacer una build para dispositivos móviles. Usando el comando *meteor add-platform Android* añadiríamos la plataforma Android al proyecto. Se puede hacer de manera similar para los dispositivos iOS de Apple, pero esto tiene unos requisitos (un ordenador Mac, una cuenta de desarrollador de Apple) de los que no dispongo en el momento de hacer la memoria. Añadir una plataforma al proyecto implica que al ejecutar el comando *meteor build*, que explicaremos en breves momentos, se genera también los ejecutables para dicha plataforma. En el caso de Android, se generará un apk adicional que podrá correr en los dispositivos Android. Ahora pasamos a ejecutar el comando *meteor build*, lo que creará los archivos necesarios para ejecutar la aplicación. Al haber añadido Android como plataforma, hemos de pasarle al comando *build* la dirección del servidor en el que estará hosteada la aplicación, para que la apk tenga una dirección y un puerto al que conectarse. Además, se genera un archivo tar que contiene la aplicación NodeJS que correrá en el servidor. Tras instalar las dependencias necesarias, y seguir los pasos detallados en el Anexo A, la aplicación estará activa y accesible, y tendremos un apk que podrá instalarse en los dispositivos Android compatibles y/o subirse a la Google Play Store para su difusión.

5.2 Pruebas

Las únicas pruebas realizadas sobre la aplicación han sido pruebas unitarias respecto a todos los módulos del servidor. Han sido pruebas manuales, que se han llevado a cabo de dos maneras: primero, mediante pruebas de caja negra llamando a los métodos, y luego usando la interfaz del cliente e intentando meter datos erróneos.

Figura 5-1: Modelo de caja negra

El método de pruebas de caja negra consiste en llamar a una función con unos parámetros y mirar si la salida de dicha función coincide con lo que debería ser. En mi opinión, es una forma muy buena de desarrollar aplicaciones, ya que podemos hacer los prototipos de las funciones, sabiendo qué debe entrar y qué debe salir, y, tras probar que las salidas son las esperadas para las entradas correspondientes, podemos descartar errores en esos métodos ahorrándonos tiempo y esfuerzo.

En cuanto a las pruebas de estrés, en el momento de hacer la memoria no se sabe la máquina en la que correría la aplicación, por lo que no son concluyentes. Uno de los aspectos negativos de Meteor, como la mayoría de las plataformas de desarrollo de aplicaciones web, es que no provee un método de realización de pruebas de este tipo, lo que requiere la creación de métodos y scripts adicionales. Aparte

6 Conclusiones y trabajo futuro

6.1 Conclusiones

Tras la realización de este TFG, es necesario comprobar que cumple los requisitos iniciales, si supone una mejora para el sistema actual y qué ha sido importante durante la realización del mismo.

El trabajo, en mi opinión, cumple los requisitos iniciales: Sirve como plataforma en la que el profesor puede llevar a cabo un seguimiento de la actividad académica de sus alumnos, pasando lista a la vez que observa el nivel de aprendizaje clase a clase. También podemos decir que se trata de un sistema sencillo y poco invasivo, que no altera el modo de vida ni de alumnos ni de profesores significativamente, lo cual, a nivel personal es un objetivo importante, ya que, como alumno, si se me hubiera presentado la oportunidad de usar una aplicación así la habría agradecido mucho.

También veo importante decir que este trabajo no vale nada sin el esfuerzo, por poco que sea, de profesores y alumnos. Un profesor que se tome tiempo en hacer cuestionarios útiles, que observe los resultados y se preocupe en modificar las clases para suplir las carencias de los alumnos de cada una (descubiertas gracias a la app desarrollada) puede afectar positiva y enormemente a los resultados de sus alumnos.

En cuanto a Meteor, la plataforma utilizada, mis conclusiones son que ha sido una elección muy acertada. Posee una amplia documentación, y emplea el lenguaje JavaScript, conocido por todos. La integración entre cliente y servidor, vistas y métodos y base de datos es muy sencilla una vez se conoce el funcionamiento general de la plataforma. Además, posee infinidad de plugins que ayudan con el desarrollo de la aplicación, como plugins de generación de gráficas a partir de datos, plugins de mensajes de alertas, etc. que hacen que la aplicación sea más sencilla de desarrollar y obtenga un aspecto visual mucho mejor de lo que se conseguiría sin empelarlos. Esto no quita que el diseño sea una parte importantísima a la hora de crear una aplicación de este estilo.

Por último, me he dado cuenta de la gran importancia de un aspecto que se recalca a lo largo de la mayoría de la carrera: cuanta más gente pruebe la aplicación en desarrollo, más fallos se encontrarán. Tras desarrollar un módulo, como puede ser la creación de un cuestionario, al desarrollador, en este caso, a mí, puede parecerle muy funcional y sencillo, pero cuando lo prueban otras personas, opinan diferente. Esta aplicación ha pasado por muchos cambios en cuanto al diseño de la aplicación se refiere, muchos de los cuales vienen de sugerencias de otras personas que, tras minutos de uso se dan cuenta de cosas esenciales, mientras que el desarrollador, tras horas de testeo, no ve. Esto demuestra la gran importancia de que no una, sino varias personas prueben las apps antes de finalizarlas, ya que el resultado final será mucho mejor.

6.2 Trabajo futuro

La aplicación desarrollada puede obtener mucha más funcionalidad. Está claro que una plataforma web de este estilo tiene infinitas posibilidades, pero en mi opinión no hay que confundirla con otras, como por ejemplo Moodle, ya que el rumbo que debería tomar es uno completamente diferente.

Es una aplicación pensada no para sustituir a sistemas como Moodle, sino para complementarlos. Por ello, funcionalidad como subir apuntes, enviar prácticas y demás quedan descartadas. Pero una funcionalidad interesante, y que sería muy útil, quitando esfuerzo a alumnos y profesores usuarios de la aplicación desarrollada sería la creación de un **middleware** que unificase de cierta manera ambas aplicaciones. Esto es, un software que, por ejemplo, crease con los usuarios y cursos de Moodle los equivalentes en esta aplicación para que al comenzar el año universitario ya estuvieran creados. Está claro que es una característica ambiciosa y que llevaría esfuerzo por parte de personal de ambas aplicaciones, pero podría obtenerse un gran resultado.

En el momento de escribir esta memoria, la aplicación solo tiene vista móvil, por lo que un objetivo futuro sería adaptarla para que también se pudiera usar cómodamente en ordenadores.

Enfocando la aplicación desde este punto de vista, podrían desarrollarse características complementarias como mensajería instantánea profesor-alumno, para preguntar dudas acerca de temas vistos en clase, o comentarios en los cuestionarios para aclarar dudas de preguntas. También se debería implementar cualquier estadística que los profesores vean importante a la hora de evaluar el conocimiento de los alumnos. Además, podrían implementarse nuevos tipos de preguntas, como preguntas de respuesta en texto, preguntas con imágenes, preguntas de ordenar de más a menos correcta… Hay muchas posibilidades de mejora y ampliación.

Referencias

[1] Meteor: https://docs.meteor.com/
[2] JQuery: https://jquery.com/
[3] StackOverflow: https://stackoverflow.com
[4] NodeJS: https://nodejs.org
[5] MongoDB: https://docs.mongodb.com/

Glosario

API	Application Programming Interface
Middleware	Software que asiste a una aplicación para interactuar o comunicarse con otras aplicaciones, o paquetes de programas, redes, hardware y/o sistemas operativos
App	Aplicación
Meteor	Framework para aplicaciones web con JavaScript libre y de código abierto escrito usando Node.js
MongoDB	Sistema de base de datos NoSQL orientado a documentos, desarrollado bajo el concepto de código abierto.
DOM	Document Object Model (Modelo de Objetos del Documento)

Anexos

A Manual de instalación

La aplicación desarrollada puede instalarse en varios sistemas operativos, pero en concreto se va a explicar cómo instalarse en un sistema operativo Linux. Vamos a diferenciar dos sistemas: el de desarrollo y el de instalación. En el ordenador de desarrollo será necesario tener instalado Meteor, que se encargará de instalar el resto de los componentes necesarios en este ordenador. En el ordenador de instalación será necesario instalar Node.js y MongoDB.

Lo primero que hay que hacer es situarse en el directorio del proyecto de Meteor que hemos desarrollado durante el TFG. Ahora es el momento de añadir las plataformas para las que se desea crear la aplicación, como Android o iOS, mediante el comando *meteor add-platform*. El añadir estas plataformas implica diversas cosas de las que avisará la consola, como por ejemplo tener la API de Android, establecer un nombre de host mediante el que el cliente accederá a la aplicación, etc. Una vez hecho esto, hay que ejecutar el comando *meteor build.* Esto generará los archivos para las diversas plataformas, y un archivo .tar que contiene la aplicación node que ha de correr en el servidor.

En el ordenador que va a hacer de servidor, una vez instalados MongoDB y Node.js, descomprimimos el .tar creado como se acaba de explicar. Se crea una carpeta llamada bundle, dentro de la cual encontramos un archivo README. Tras seguir las instrucciones de dicho archivo, la aplicación quedará instalada, y para ponerla en funcionamiento tan solo hace falta ejecutar el comando *node main.js*.

B Manual del programador

La aplicación emplea variables de entorno en múltiples situaciones. Para ajustar una variable de entorno, basta con ejecutar el comando *export <nombre_variable> = <valor_variable>*. Antes de ejecutar la aplicación es necesario ajustar algunas variables de entorno:

- **MONGO_URL**: URL de la base de datos MongoDB que empleará la aplicación,
- **ROOT_URL**: URL de acceso a la aplicación. Se empleará en el link de los correos que se envían para recuperar la contraseña o para verificar la cuenta.
- **PORT**: Puerto interno en el que la aplicación se ejecutará.
- **MAIL_URL**: URL del servicio SMTP que se empleará para enviar los emails.

Cuando se quiera añadir un usuario con el rol de profesor a la aplicación, basta con introducir su dirección de correo en la base de datos. La consulta MongoDB para hacerlo es la siguiente:
*db.lista_profesores.update({_id: "unique"}, {$addToSet: {emails: "**nombre@email.com**"}})*

Cuando se quiera añadir un usuario con el rol de profesor a la aplicación, basta con introducir su dirección de correo en la base de datos. La consulta MongoDB para hacerlo es la siguiente:
*db.lista_alumno.update({_id: "unique"}, {$addToSet: {sufijos: "**email.com**"}})*